HBnB Evolution - Technical Documentation

Authors: Raghad, Rama, Jana
Date: February 5, 2026
Version: 1.0

---

1. Introduction

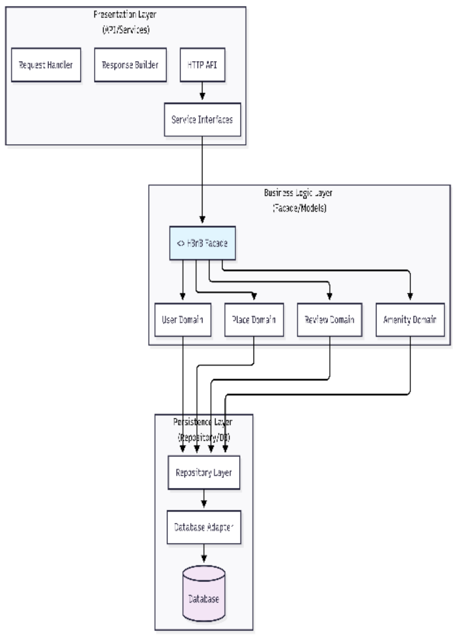
HBnB Evolution is a simplified AirBnB-like platform connecting property owners with guests. Users can list properties, search for accommodations, and submit reviews.

Purpose: Blueprint for development team
Scope: Architecture, entities, API flows, design decisions

---

2. High-Level Architecture

Three-Layer Design:



Presentation Layer

- **Components:** API endpoints (UserAPI, PlaceAPI, ReviewAPI, AmenityAPI)
- **Functions:** HTTP handling, validation, authentication, response formatting

Business Logic Layer

- **Components:** HbnbFacade (Facade Pattern), Domain Models
- **Functions:** Business rules, validation, entity coordination

Persistence Layer

- **Components:** Repositories, Database (PostgreSQL/MySQL)
- **Functions:** Data storage, CRUD operations, queries

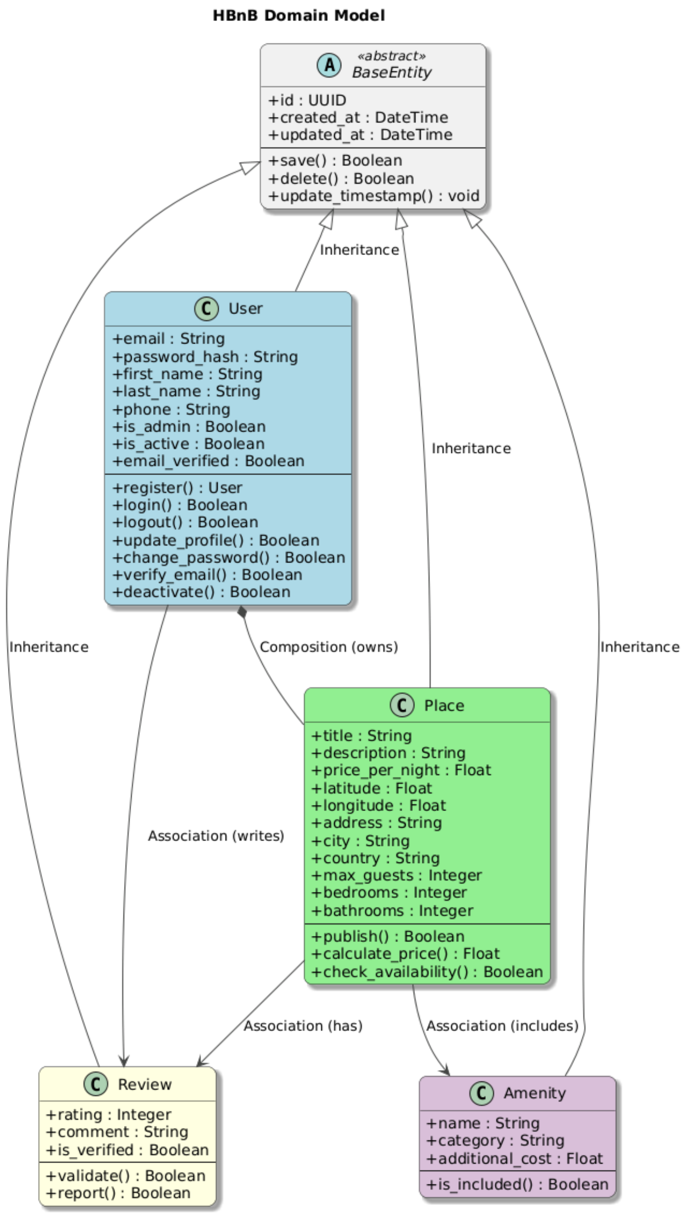
Facade Pattern: Simplifies API → Business Logic interaction

- Without: API → Service → UserManager → PlaceManager → Repository
- With: API → Service → HbnbFacade → [Internal] → Repository

---

3. Business Logic Layer

Core Entities



BaseEntity (Abstract)

- id: String (UUID4 format)
- created_at: DateTime
- updated_at: DateTime

User

- first_name, last_name, email, password_hash, is_admin: Boolean
- Rules: Unique email, hashed password, admin flag

Place

- title, description, price: Float, latitude: Float, longitude: Float, owner_id, amenity_ids
- Rules: Price > 0, valid coordinates, owner verification

Review

- rating: Integer, comment, user_id, place_id
- Rules: Rating 1-5, one review per user per place

Amenity

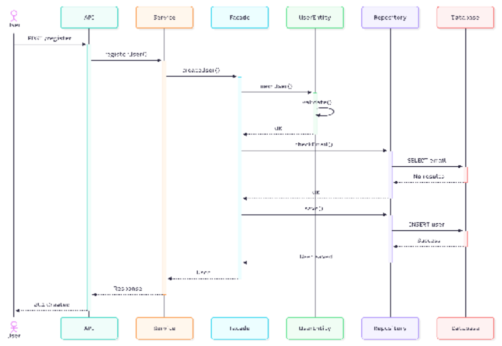
- name, description
- Rules: Unique names, admin-only management

Relationships

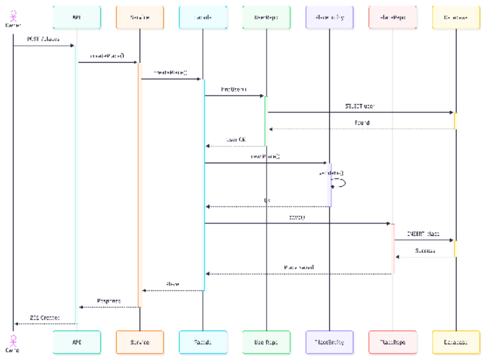
1. **User → Place** (1-to-many): One user owns many places
2. **User → Review** (1-to-many): One user writes many reviews
3. **Place → Review** (1-to-many): One place receives many reviews
4. **Place ↔ Amenity** (many-to-many): Places have multiple amenities

4. API Interaction Flow

User Registration

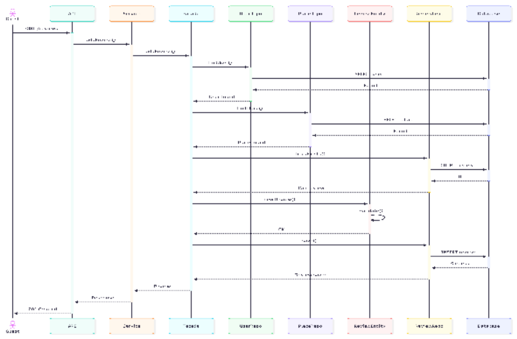


Flow: Client → API → Service → Facade → User → Repository → Database
Key: Email uniqueness check, password hashing, UUID4 generation



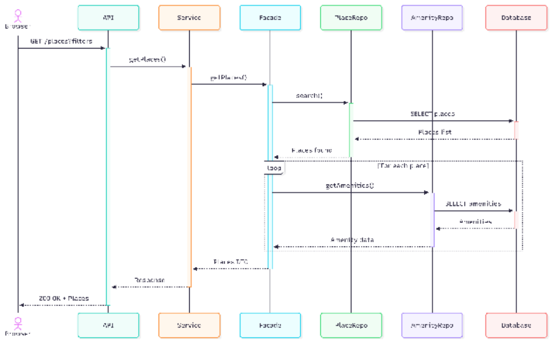
Place Creation

Flow: Authenticated user → API → Facade → Place → Repository
Key: Owner verification, coordinate validation, amenity association



Review Submission

Flow: Authenticated user → API → Facade → Review → Repository
Key: Duplicate check, rating validation (1-5), user-place verification



Fetch Places List

Flow: Client → API → Facade → Repository → Database → Enrichment
Key: Filter processing, pagination, amenity data enrichment

5. Design Decisions

1. **Three-Layer Architecture** - Separation of concerns, scalability.
2. **Facade Pattern** - Simplified API layer, centralized coordination.
3. **UUID4 IDs** - Global uniqueness, security, database independence.

4. **Boolean Admin Flag** - Simple meets requirements, easy to extend.
5. **Repository Pattern** - Database abstraction, testability.
6. **Multi-level Validation** - Defense in depth, data integrity.

---

6. Implementation Guidelines

Phases

1. **Foundation:** BaseEntity, database schema, repositories.
2. **Core Entities:** User, Place, Review, Amenity, Booking implementations.
3. **Business Logic:** Facade, validation, transaction management.
4. **API Layer:** Endpoints, authentication, error handling.
5. **Testing:** Unit, integration, end-to-end tests.

Security

- JWT authentication.
- Password hashing (bcrypt).
- SQL injection prevention.
- Input validation.

Performance

- Database indexing.
- Query optimization.
- Catching strategies.
- Pagination for large datasets.

---

7. Conclusion

Key Achievements:

- Clear three-layer architecture with Facade pattern.
- Comprehensive domain model with business rules.
- Well-defined API interaction patterns.
- Scalable and maintainable design.